Práctica 3. Divide y vencerás

Francisco Javier Molina Rojas
javier.molinarojas@alum.uca.es
Teléfono: 722528757
NIF: 45386606Q

17 de diciembre de 2022

1. Describa las estructuras de datos utilizados en cada caso para la representación del terreno de batalla.

   La estructura utilizada sera un vector que contendra cada una de las celdas del mapa. Ademas una celda contendra su posicion ademas de la valoración para dicha celda.

Estructura de Celda

```
struct Celda
{
        int row,col;
        float valor;
        Celda(int r, int c, int v): row(r), col(c), valor(v) {}
        Celda(): row(0),col(0),valor(0) {}
};
```

Vector de Celdas

```
        std::vector<Celda> Celdas;
        for(int i = 0 ; i < nCellsWidth ; i++)
        {
                for(int j = 0 ; j < nCellsHeight ; j++)
                {
                        //insertamos todas las celdas en la lista de candidatos. Para poder ordenarla luego usaremos
                                el metodo sort
                        Celdas.push_back(Celda(i,j,defaultCellValue(i,j,freeCells,nCellsWidth,nCellsHeight,mapWidth,
                                mapHeight,obstacles,defenses)));
                }
        }
```

2. Implemente su propia versión del algoritmo de ordenación por fusión. Muestre a continuación el código fuente relevante.

Algoritmo de Fusión

```
std::vector<Celda> fusion(std::vector<Celda>& FirstHalf,std::vector<Celda>& SecondHalf){

        std::vector<Celda> Results(FirstHalf.size() + SecondHalf.size());// Vector resultado (Vector ordenado creado
                por la fusion entre la primera mitad y la segunda)
        int i = 0, j = 0, k = 0;//Variables necesarias para recorrer cada vector



        while(i < FirstHalf.size() && j < SecondHalf.size()) //Mientras que no hayamos recorrido uno de los vectores
                enteros
        {
                if(FirstHalf[i] < SecondHalf[j]) //si la posicion i de la primera mitad es menor que la posicion j
                        de la segunda mitad
                {
                        Results[k] = FirstHalf[i];      //Asignamos el elemento en resultado
                        i++;//iteramos i para la primera mitad
                }
                else
                {
                        Results[k] = SecondHalf[j];//Asignamos el elemento en resultado
                        j++; //iteramos j para la segunda mitad
                }
                k++; //iteramos k para el resultado
        }
        //Rellenar lo faltante de cada mitad
        while(i < FirstHalf.size())
        {
                Results[k] = FirstHalf[i];
                i++;
                k++;
        }
        while(j < SecondHalf.size())
```

```
        {
                Results[k] = SecondHalf[j];
                j++;
                k++;
        }
        return Results;
}


void ordenacionFusion(std::vector<Celda>& v)
{
    int n = v.size()/2; //obtenemos la posicion de la mitad del vector

    if(n > 0)    //si es mayor de 0
    {
        std::vector<Celda> FirstHalf(n); //creamos 2 vectores auxiliares
        std::vector<Celda> SecondHalf(v.size() - n);// Uno guardara la primera mitad y otro la segunda

        for(int i = 0; i < n; i++)       //relleno de la primera mitad
        {
                FirstHalf[i] = v[i];
        }

        for(int i = n ; i < v.size(); i++) //relleno de la segunda mitad
        {
                SecondHalf[i-n] = v[i];
        }
        ordenacionFusion(FirstHalf); //llamada recursiva con la primera mitad
        ordenacionFusion(SecondHalf);//llamada recursiva con la segunda mitad
        v = fusion(FirstHalf,SecondHalf); //llamada a funsion
    }
}
```

3. Implemente su propia versión del algoritmo de ordenación rápida. Muestre a continuación el código fuente relevante.

Algoritmo Rapido

```
int pivote(std::vector<Celda>& v,int i, int j)
{
        int pivot = i;
        int x = v[i].valor;

        for(int k = i +1 ; k <= j ; k++)
        {
                if(v[k].valor <= x)
                {
                        pivot++;
                        Celda aux = v[k];
                        v[k] = v[pivot];
                        v[pivot] = aux;
                }
        }

        v[i] = v[pivot];
        v[pivot].valor = x;
        return pivot;
}

void Quicksort(std::vector<Celda>& v,int i, int j)
{
        int n = j-i+1; //obtenemos el numero de elementos del rango del vector

        if(n > 0)       //si es mayor de 0
        {
                int pivot = pivote(v,i,j); //obtener pivote
                Quicksort(v,i,pivot-1); //recursividad
                Quicksort(v,pivot+1,j);
        }
}
```

4. Realice pruebas de caja negra para asegurar el correcto funcionamiento de los algoritmos de ordenación implementados en los ejercicios anteriores. Detalle a continuación el código relevante.

Algoritmo Caja Negra

```
void CajaNegraFusion(std::vector<Celda>& v)
{
        ordenacionFusion(v); //realizamos ordenacion
        bool bienordenado = true;
        for(int i = 0; i < v.size()-1 ; i++)
        {
                if(v[i+1] < v[i])
                {
                        bienordenado = false;
                }
        }
}

void CajaNegraQuickSort(std::vector<Celda>& v)
{
        Quicksort(v,0,v.size()-1);
        bool bienordenado = true;
        for(int i = 0; i < v.size()-1 ; i++)
        {
                if(v[i+1] < v[i])
                {
                        bienordenado = false;
                }
        }
}
```

5. Analice de forma teórica la complejidad de las diferentes versiones del algoritmo de colocación de defensas en función de la estructura de representación del terreno de batalla elegida. Comente a continuación los resultados. Suponga un terreno de batalla cuadrado en todos los casos.

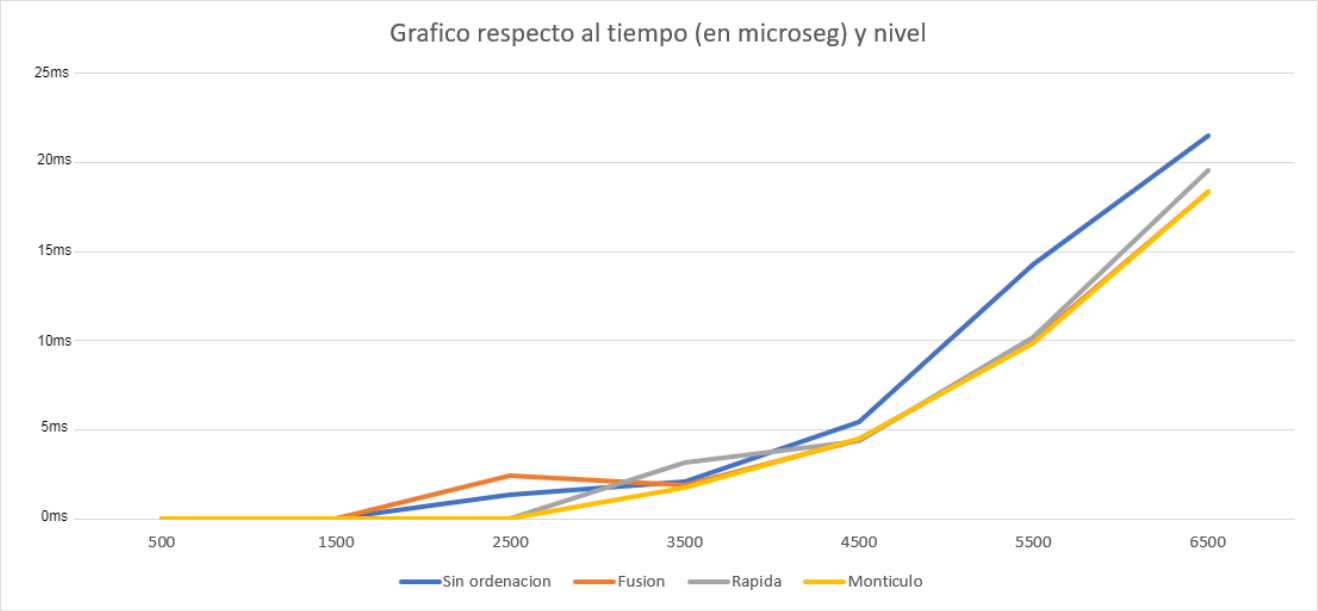
   En el caso de no realizar ordenacion, el algoritmo de selección tendría un coste de $O(n^2)$.

   En el caso de la preordenación por fusion, el algoritmo de selección tendría un coste en el peor caso de O(nlogn).

   En el caso de la preordenación rapida, el algoritmo de selección tendría un coste en el peor caso de $O(n^2)$.

   En el caso de la preordenación por montículo, el algoritmo de selección tendría un coste de O(nlogn) (debido a la creacion del monticulo y luego a la operacion pop).

6. Incluya a continuación una gráfica con los resultados obtenidos. Utilice un esquema indirecto de medida (considere un error absoluto de valor 0.01 y un error relativo de valor 0.001). Es recomendable que diseñe y utilice su propio código para la medición de tiempos en lugar de usar la opción *-time-placeDefenses3* del simulador. Considere en su análisis los planetas con códigos 1500, 2500, 3500,..., 10500, al menos. Puede incluir en su análisis otros planetas que considere oportunos para justificar los resultados. Muestre a continuación el código relevante utilizado para la toma de tiempos y la realización de la gráfica.



Todo el material incluido en esta memoria y en los ficheros asociados es de mi autoría o ha sido facilitado por los profesores de la asignatura. Haciendo entrega de este documento confirmo que he leído la normativa de la asignatura, incluido el punto que respecta al uso de material no original.